In the Specification:

On page 1, before paragraph [0001], please insert:

This application is a Continuation of prior U.S. Patent Application Serial No. 09/728,168 filed December 1, 2000 entitled "System and Method to Store Data in the System Memory of a Computing Device." The application is expressly incorporated herein, in its entirety, by reference.

Please amend paragraphs [0016] through [0019] in the following manner:

[0016] In an exemplary embodiment of the present invention, system memory 110 is not treated as one big chunk of memory space (e.g., 128 Mbytes) to be allocated for data at random, but rather is divided for more efficient use by processor 120. The division of system memory 110 is based on the principle that data of the same size will be stored in ~~the same continuos areas~~ a continuous area of system memory 110. This data will not become fragmented because as data is erased from system memory 110, any new data of the same size may be written into the memory space vacated by the erased data.

[0017] Fig. 3 shows an exemplary division of system memory 110 into various size chunks that are grouped into chunk pools according to the present invention. A chunk is a continuous physical memory space of a specified size. These are the continuous memory areas where data of the same size may be stored. Chunks 151-152 may have a size of, for example, 512 bytes each and belong to chunk pool 150, chunks 161-162 are twice as large at 1,024 bytes each and belong

to chunk pool 160, and likewise, in this exemplary embodiment each of the successive chunks in the remaining chunk pools are twice as large as the chunks in the previous chunk pools. Thus, chunks 171-172 are 2,048 bytes, chunks 181-182 are 4,096 bytes and chunks 191-192 are 8,192 bytes. All chunks are memory spaces within system memory 110 and may be located anywhere throughout system memory 110; the only requirement ~~may be~~ is that the bytes in a particular chunk are continuous. For example, as described above, system memory 110 may have an overall size of 128 Mbytes which could be represented as ~~128,000,000~~ 2^{27} consecutive storage areas having 1 byte of storage. Therefore, chunk 151 having a size of 512 bytes may occupy any 512 bytes within the 128 Mbytes, but the 512 bytes occupied by chunk 151 should be continuous. Thus, chunk 151 may occupy byte locations 1-512 in system memory 110 or, just as likely, it may occupy locations 10,001-10,512 or any other continuous memory area of 512 bytes. Chunk pools ~~150-190~~ 150, 160, 170, 180 and 190 manage the memory chunks of a certain fixed predetermined size. For example, chunk pool 150 manages chunks ~~pools~~ 151-152 which ~~has~~ have a size of 512 bytes. If any additional chunks of 512 bytes are created, such a new chunk would be managed by chunk pool 150 because it has been defined as the manager of 512 byte chunks. The defining of chunks and chunk pools will be described in greater detail below.

[0018] The number of chunks and chunk pools and the sizes of each described with reference to Fig. 3 are only exemplary and any number of chunks and/or chunk pools of any size may be selected in implementing the present invention. Additionally, there is no requirement that each of the chunks be one half the size of the next largest chunk or that the larger chunks be some factor of the smaller chunks. However, as will be described in greater detail below, there may be

some embodiments of the present invention which are more easily implemented when the larger chunks are some factor of the smaller chunks as shown in Fig. 3. It should also be noted that the chunks ~~151-192~~ shown in Fig. 3 do not need to exist at the time system 100 is started. The chunks may be dynamically created as they are needed during runtime. This dynamic creation of chunks will be described in more detail below.

[0019] In addition to chunks and chunk pools, an exemplary embodiment of the present invention also includes units and unit pools. Units are the continuous portion of memory which are the size of the data that each unit pool will store. As described above, data of the same size will be stored in the same continuos areas of system memory 110. Unit pools manage units of a certain fixed predetermined size. Each unit pool will only manage units of the same size. Fig. 4 shows an exemplary relationship between units, unit pools, chunks and chunk pools according to the present invention. Fig 4 shows chunk pool 150 which manages chunks having sizes of 512 bytes. At the present moment there are two 512 byte chunks 151-152 being managed by chunk pool 150. There are also two defined unit pools 200 and 210. Unit pool 200 may be assigned to manage units having a size of four bytes. Thus, the units of unit pool 200 may be assigned as the storage location for data with a size of four bytes. Unit pool 210 may be assigned to manage units having a size of eight bytes. Thus, the units of unit pool 210 may be assigned as the storage location for data with a size of eight bytes. However, unit pools 200 and 210 cannot have any actual units until there is a physical storage location to store the associated data. Therefore, a unit pools is associated with a chunk pool where the unit pool may assume control over one or more of the chunks controlled by the chunk pool. Once the unit pool assumes control of a chunk,

it may separate the chunk into a number of units corresponding to the unit size for which the particular unit pool is responsible. Then, the unit pool has actual physical storage locations for the data.

Please amend paragraph [0021] in the following manner:

[0021] When all units 201-206 of chunk 151 are full, there may be another chunk allocated for the storage of four byte data sets. The process for allocating a new chunk from chunk pool 150, when it does not have any immediately available will be described in greater detail below. However, this new allocation will only take place when chunk [51] 151 is completely full. For example, if the first four byte data set stored in unit 201, is no longer needed by the software program, this data is erased from unit 201 which then becomes available for use by another four byte data set. In this case, when the software program needs to store another four byte data set, unit 201 is now available to store this new four byte data set.

Please amend paragraph [0023] in the following manner:

[0016] Fig. 5 shows an exemplary process flow for implementing a memory manager according to the present invention. As described above with reference to Fig. 2, during the operation of system 100, processor 120 may desire to move data from permanent memory storage device 130 to system memory 110. The exemplary memory manager process described with respect to Fig. 5 is directed to the orderly storage of this data in the system memory. The exemplary process of Fig. 5 may be carried out by a set of instructions that are stored in, for example, permanent memory storage device 130 or system memory 110 and executed by, for example, processor 120. Those skilled in the art will understand that system 100 may include multiple other devices than those depicted in Fig. 2, and the memory manager function may be stored and/or executed by any of these other devices. Starting with step 305, the chunk pools and chunks are defined, for

example, the chunk pools ~~150-190~~ and chunks ~~151-192~~ described with reference to Fig. 3. It is not necessary to allocate the system memory space for the chunks, just to define the various sizes of the chunks and the chunk pools which the memory manager will support. The allocation of the memory space for the defined chunks may take place at a later time during system operation when the particular chunk is needed. It may also be possible to allocate the memory space for the chunks when the chunks are defined in step 305. In this exemplary embodiment of the present invention, memory space from system memory will be allocated for one chunk in step 310. The one chunk that will have system memory space allocated will be one of the chunks that is the largest chunk size defined in step 305. For example, if in step 305, chunks are defined that have the sizes of the chunks ~~151-192~~ described with reference to Fig. 3, that is chunks of 512 bytes, 1,024 bytes, 2,048 bytes, 4,096 bytes and 8,192 bytes. ~~In~~, in this example, in step 310, one chunk of 8,192 bytes in chunk pool 190, the largest defined size will be allocated from system memory. The reason for initially allocating this one chunk is to prevent a back and forth or "ping-pong" effect between the chunks and system memory.

Please amend paragraphs [0029] through [0031] in the following manner:

[0029] Fig. 6 shows exemplary chunks ~~400-420~~ 400, 410 and 420 under the control of unit pools ~~430-450~~ 430, 440 and 450 and associated units storing data as described in step 350 of the exemplary process of Fig. 5. Each of chunks 400, 410 and 420 have been previously defined in terms of size and terms of the size of the units with which they are associated as in, for example, steps 305 and 315 of Fig. 5, respectively. Chunk 400 is an exemplary 32 byte chunk which is associated with 4 byte units 401-408. Chunk 410 is an exemplary 64 byte chunk which is associated with 8 byte units 411-418. Chunk 420 is an exemplary 128 byte chunk which is associated with 16 byte units 421-428. Four byte units 401-403 of chunk 400 already have data stored in them, thus pointer 409 is directing that the next piece of data that is four bytes should be stored in unit 404. For example, referring back to Fig. 5, if in step 320 it is determined that the data size is four bytes, the process continues to step 325 which determines that four bytes is not greater than the maximum unit pool size. The process continues to step 335 which determines

that unit pool 430 is responsible for four byte data sets. In step 340 it is determined that there is an available unit of four bytes in chunk 400 of unit pool 430 by locating pointer 409 which is pointing at vacant four byte unit 404. The process then goes to step 350 which stores the current four byte data into unit 404 and pointer 409 advances to unit 405 indicating that the next piece of four byte data should be stored in unit 405.

[0030] As described above, each chunk is a continuous memory area in the system memory 110. Thus, 32 byte chunk 400 is allocated 32 continuous bytes of system memory 110 and each of the units 401-408 associated with chunk 400 are continuous within chunk 400. For example, unit 401 may occupy bytes 1-4 of chunk 400, unit 402 may occupy bytes 5-8 of chunk 400, and so on, until all 32 continuous bytes are accounted for by units. Therefore, when data is stored in a particular unit the pointer information is given to processor 120 and when processor 120 wants to retrieve the data, processor 120 only needs to refer to the pointer information, without any additional information. Processor 120 does not need to know the size of the data because the mere fact that the pointer information is directing processor 120 to a four byte unit pool indicates that the data is four bytes. Additionally, there is no need to set a bit in the data that signifies that it is in the last byte of the data because processor 120 will only take a single unit at a time. For example, if processor 120 wanted to retrieve the four byte instruction stored in unit 402, the pointer to unit 402 would be stored by processor 120 when the data was stored in unit 402. The pointer information would indicate to processor 120 that unit 402 was a four byte unit starting at a particular byte in system memory, *e.g.*, the fifth byte of chunk 400. Processor 120 then accesses this location and retrieves that byte and the following three bytes to gain access to the entire instruction stored in unit 402. If the data was not stored continuously, then a four byte piece of data may require four separate pointers to various physical locations in system memory including an indication of how to reassemble the bytes.

[0031] Continuing with Fig. 6, it can be seen that all of units 411-418 of chunk 410 have been filled with eight byte data. Therefore, there is no pointer pointing to any of units 411-418 because they are not available to accept data. Once again, referring back to Fig. 5, and

considering that it is determined that the piece of data is eight bytes in step 320 and that there are defined unit pools capable of storing data of this size in step 325. ~~In~~ in step 335 it is determined that unit pool 440 is responsible for eight byte data sets. Proceeding to step 340, if chunk 410 is the only current chunk that has memory space allocated for eight byte data, then the process would go to step 345 because there are no currently available units to accommodate data with the size of eight bytes, indicated by the lack of a pointer to an empty unit in chunk 410. Units 421, 422 and 424-428 of chunk 420 are filled with sixteen byte data. However, unit 423 is vacant and pointer 429 is indicating that the next piece of sixteen bit data should be stored in unit 423. The vacancy in unit 423 may be caused by processor 120 erasing data from this unit as a result of, for example, the closing of a network connection or an application program. Once a unit is vacated, it immediately becomes available for another piece of data of the same size. Additionally, all the units of a chunk pool should be filled before allocating memory space for another chunk pool having the same size unit pools. If a first chunk of a unit pool is filled and a second chunk becomes part of the same unit pool, and subsequently, a unit of the first chunk is erased, the pointer will continue to fill the second chunk to avoid the "ping-pong" effect between the first and second chunks. However, when the second chunk is filled the pointer will go back to the first chunk, rather than allocating a new chunk to the unit pool.